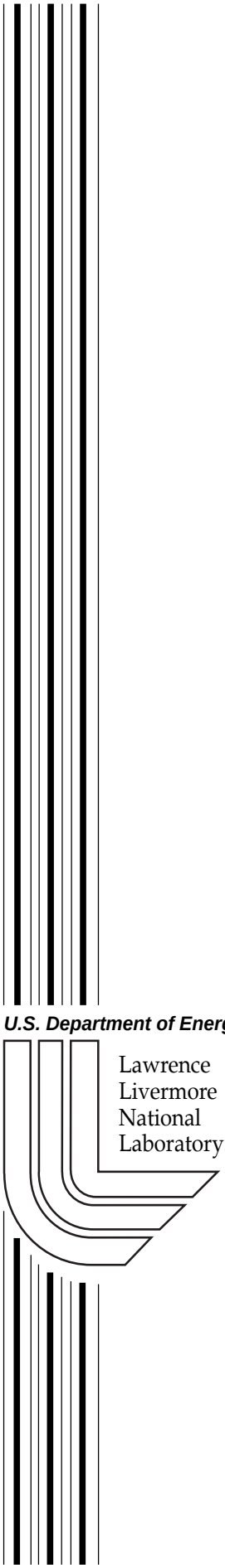
Preprint
UCRL-JC-152962

On Issues of Precision for Hardware-Based Volume Visualization

E. LaMar

This article was submitted to: *IEEE Parallel Visualization and Graphics Symposium, Seattle, WA, USA, 10/19/2003 – 10/24/2003*



April 11, 2003

This work was performed under the auspices of the United States Department of Energy by the University of California, Lawrence Livermore National Laboratory under contract No. W-7405-Eng-48.

On Issues of Precision for Hardware-based Volume Visualization

Eric LaMar

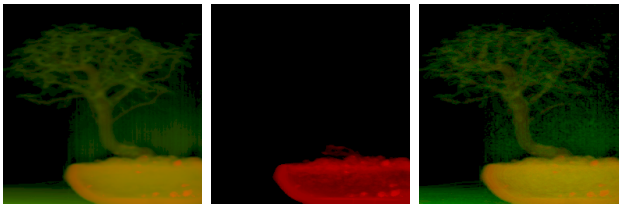
11th April 2003

Abstract

This paper discusses issues with the limited precision of hardware-based volume visualization. We will describe the compositing OVER operator and how fixed-point arithmetic affects it. We propose two techniques to improve the precision of fixed-point compositing and the accuracy of hardware-based volume visualization. The first technique is to perform dithering of color and alpha values. The second technique we call *exponent-factoring*, and captures significantly more numeric resolution than dithering, but can only produce monochromatic images.

keywords: volume visualization, dithering, numerical techniques, digital compositing

1 Introduction



Software Hardware Dithering

Figure 1: A comparison software, hardware, and hardware with dithering rendering techniques on the bonsai dataset.

Graphics-hardware based volume visualization is capable of significantly faster visualization of a dataset than software based techniques. However, graphics hardware based techniques have a significant issues with image accuracy because there is insufficient numeric precision to properly implement the compositing math. While prior works have noted that the limited depth of frame buffers impacts image accuracy, there has no other discussion and characterization of the limited precision and what can be done to mediate it. Figure 1 compares our technique to software and naive hardware solutions, and shows that our technique is capable of capturing most of the important image features.

Hardware-based volume visualization techniques, including cell-projection, slicing, and textures all exhibit the artifacts and issues discussed in this paper, so the results of the paper are applicable to any rendering technique that uses limited-precision, fixed-point arithmetic to implement the OVER operator.

The OVER operator came from computing overlay matting for cartoon animation. While fine for that purpose, the limitation of graphics hardware do not permit scaling of the OVER operator.

We use per-unit-distance opacity because reproducing the same image for datasets of different spatial resolutions and extents is difficult using per-voxel opacity but trivial using per-unit-distance opacity. We classify three forms of error resulting from a fixed-point implementation of the OVER operator: *representation*, *pigeon-hole*, and *weight-rounding*.

We describe two techniques to reduce these errors. Our first technique is to dither the per-voxel color and opacity values in the spatial domain. That is, for a series of voxels whose per-voxel value cannot be accurately represented, we periodically round the value up and down to ones that can be represented. The aggregate effect of compositing those voxels is to approximate the original per-voxel value. The second we call *exponent-factoring*, where the exponent of the per-unit-distance to per-voxel opacity transformation is incrementally applied. The data space is decomposed hierarchically using an N-ary tree and performing the compositing using local per-unit-distance and per-voxel opacities. This technique captures significantly more detail of a large volume.

2 Related Work

Hardware compositing started as a technique to automate and accelerate the process of cartoon animation [Wal81, PD84]. A "limited precision" OVER operator is fine for cartoon animation because the number of layers tends to be small (less than 10) and partial transparency is used to represent sub-pixel occlusion and not "real" transparency. There are few occasions that these partially transparent regions are repeatedly composited on top of each other.

Cabral *et.al.* [CCF94] discuss how hardware-accelerated texturing can perform volume rendering and how this technique can also be used to reconstruct a volume from tomographic image set.

Kim *et.al.* [KWP01] and Meisserner *et.al.* [MHB+00] have compare the imagery and artifacts of different rendering modalities and have noted the differences between

techniques, but did not investigate *why* limited precision, fixed-point techniques degrade image quality. Wittenbring *et.al* [WMG98] solve the problem of color bleeding in volume visualization, but experienced poorer image quality because the colors values became too small to represent in 8 bit framebuffers.

Engel *et.al.* [EKE01] introduce a hardware-based volume rendering technique that deals with post-classification sampling rates, but does not deal with the numeric precision issues associated fixed-point quantities.

Prior research assumed that the mechanism to composite a small number of cartoon layers could be used to composite large numbers of highly transparent layers. Also, many works use a bimodal transfer-function (*i.e.*, data values become either totally opaque or totally transparent) or use volumes that are small enough to masking the artifacts of large, semi-transparent volumes. Our work explores the detailed processes the OVER operator and what can be done to improve the quality of limited precision, fixed-point based volume visualization.

3 Per-Unit Opacity vs Per-Voxel Opacity

We first discuss the relationship of per-unit-distance and per-voxel opacity in volume visualization. While others have developed formula similar to this, we feel that our approach to calculating per-voxel opacity is clearer and aids in simplifying the discussion of the genesis of image errors. It is useful to discuss transfer functions in terms of unit values, rather than voxels, so one can apply a transfer function to different datasets that contain similar data, but have different spatial size and resolution. For example, if one models a material that has an optical absorption of 50% over one inch, a user shouldn't have to calculate the per-voxel opacity if that volume is represented at a regular grid of, say 128^3 or 512^3 voxels, or, in the case of multiresolution volume visualization, a series of volume from 64^3 up to 2048^3.

From the OVER operator definition [PD84], the final color C_n and opacity Γ_n of a volume are computed, the iterative form is:

$$\begin{aligned} C_n &= \sum_{i=1}^{n} \left(c_i \alpha_i \left(\prod_{j=i+1}^{n} (1-\alpha_j) \right) \right) &+ C_0 \left(\prod_{j=1}^{n} (1-\alpha_j) \right) \\ \Gamma_n &= \sum_{i=1}^{n} \left(\alpha_i \left(\prod_{j=i+1}^{n} (1-\alpha_j) \right) \right) &+ \Gamma_0 \left(\prod_{j=1}^{n} (1-\alpha_j) \right) \end{aligned}$$

foreground background

Where c_i and α_i are the per voxel color and alpha values at sample i, and C_0 is the background color, Γ_0 is the background opacity, C_n is the final color C_n, and Γ_n is the final opacity. If we ignore the color and opacity contributions of the volume (foreground), we obtain the extinction weighted background color and the extinction value of the whole volume:

$$\begin{aligned} C_n &= C_0 \left(\prod_{j=1}^{n} (1-\alpha_j) \right) \\ \Gamma_n &= \Gamma_0 \left(\prod_{j=1}^{n} (1-\alpha_j) \right) \end{aligned}$$

From this one can see that C_n is actually the *weighted* color value, $C_n = C'_n \alpha'$, weighted by the overall opacity of the volume, which we will call α'. We note that is this representation of C_n that is difficult: it is weighted by opacity, which is often very small, so C_n is very small and prone to the representation problems. If we rewrite the equations and factor out C'_n, we obtain: $1-\alpha' = \prod_{j=1}^{n} (1-\alpha_j)$. If we assume that the material is homogeneous, $1-\alpha' = (1-\alpha)^n$. Finally, solving for the per-voxel component:

$$\alpha = 1 - (1-\alpha')^{\frac{1}{n}}. \tag{1}$$

If one solves for the voxel opacities for two differing sample depths, say a and b, one obtains: $\alpha_b = 1-(1-\alpha_a)^{\frac{a}{b}}$. Most generally, given a per unit distance opacity, α', and some fractional distance, d, the opacity over that distance is

$$\alpha_d = 1 - (1-\alpha')^d. \tag{2}$$

We define our tests over an idealized unit volume, which is sampled and projected to a screen of some arbitrary resolution, where the horizontal component is X and vertical is Y. The "sample depth" corresponds to the Z component, and is the number of samples composited in the Z direction. Hence the X & Y resolutions do not affect per-voxel opacity, but the Z resolution, or sample depth, does. Sample depth, for volume visualization applications, is the number of sample points or slicing planes through a volume.

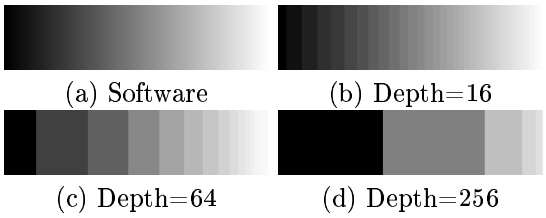
4 Problem Characterization

There are three basic issues with the lack of precision in application of the OVER operator, which we describe in the next three subsections. We use variations of a synthetic "ramp" dataset for these discussions. This is a 256×1 dataset where the voxel values vary linearly from 0 to 255 with the X axis (the Y axis size is arbitrary). The dataset has a depth of one unit, so to test different physical depths, we compute the per-voxel opacity based on the sample depth S. In the following tests, the transfer function is constructed to set the voxel color to white and the desired per-unit opacity varies linearly from 0.0 on the left to 1.0 on the right; the desired opacity for pixel (x, y) is $\frac{x}{255}$, thus the per-voxel opacity is $1-(1-x/255)^{1/S}$.

This dataset was constructed to test a "worst-case" scenario of volume rendering: compositing the same value upon itself multiple times. All of the artifacts discussed here can be difficult to see when using random or complex data. This "worst-case" scenario can be constructed with any dataset and a poorly constructed transfer function - one that maps many different data values to the same color or opacity - which then "flattens" a region and result in this form of compositing problem.

4.1 Limited Precision

The frame buffer of graphics cards stores the accumulated results over a series of compositing operations. Even if the graphics card is capable of computing intermediate results with higher precision, only the portion that fits into the frame buffer will be retained for the next composite cycle. For most cards, there are 8 bits allocated to each color channel, and cannot accurately represented the per-voxel opacity values. For example, if the desired per-unit-opacity is 0.5, the per-voxel for a sample depth of 64 is (from equation 1): $1-(1-0.5)^{\frac{1}{64}} = 0.0107719868_{10}$ or in fixed-point (truncated to 32 bits): $.00000010\ 11000001\ 11110011\ 11110011_2$. However, with only 8 bits available, only the upper 8 bits are used, which is $.00000010_2$, or 0.00781250_{10}. If we composite the value 64 times (and *not* accounting for other errors covered in this paper): $1-(1-0.00781250)^{64} = 0.3946591$, which is quite far from our desired opacity of 0.5.



(a) Software (b) Depth=16

(c) Depth=64 (d) Depth=256

Figure 2: A comparison of actual vs. desired per-unit-distance opacity using software (figure a) and hardware (figures b, c, & d).

Our experiments illustrate this error and how it can manifest itself in an image. Figure 2 shows the affects of per-voxel opacities for sample depths of 16, 64, and 256, and should form a linear ramp. Figure 2(a) show the correct image generated in software and is the baseline image. Figure 2(b) correspond to hardware compositing with a sample depth of 16 (*i.e.*, $n = 16$ in equation 1). Notice how the image and curve show regular small jumps, and is fairly close to the baseline (software) image.

Figure 2(c) shows hardware compositing with a sample depth of 64. This is a fairly poor approximation of a linear ramp. The "knees" actually separate plateau regions where the fixed-point representation is not capable of differentiating the per-voxel opacities. For example, the first plateau corresponds to compositing a value of

PUDo	PVo		Co
0.0000	0.000000	.00000000 00000000	0.0000000
0.1250	0.002084	.00000000 10001000	0.0000000
0.2500	0.004485	.00000001 00100101	0.2215804
0.3750	0.007317	.00000001 11011111	0.2215804
0.5000	0.010772	.00000010 11000001	0.3946590
0.6250	0.015209	.00000011 11100100	0.5297210
0.7500	0.021428	.00000101 01111100	0.7170170
0.8750	0.031969	.00001000 00101111	0.8689160
1.0000	1.000000	.11111111 11111111	1.0000000
(a)	(b)	(c)	(d)

Table 1: A comparison of desired per-unit-distance opacity ("PUDo", column a) vs. computed opacity ("Co", column d) for a sample depth of 64. The per-voxel opacity ("PVo", columns b and c) is shown in base-10 and base-2. Only the values in left side of column (c) can be represented in the framebuffer, so anything else is simply lost. Column (d) shows the value (in decimal) in the buffer after all compositing is completed. This table is shown graphically in Figure 2(c).

"1" 64 times. That is, all of the per-unit-distance values translate to a per-voxel value of 1 to just less than 2, but are truncated to 1. Table 1 shows this affect for a series of per-unit-distance values.

Figure 2(d) shows hardware compositing with a sample depth of 256, and is an extremely poor approximation of the desired linear ramp. As the sample depth increases, the per-voxel opacities decrease, and become more difficult to differentiate using fixed-point values.

4.2 Pigeon-Hole rounding

The second error we call the *pigeon-hole* artifact. Mathematically, the OVER operator, given two operands, produces a new value. However, then compositing a small value onto a larger value (or large value on a small one), small differences in the values will not result in different outputs.

Table 2 shows this affect using 4-bit fixed point arithmetic. Here we show all combination of composited pairs of values and the result $r = OVER(f, b)$: given a background value b (reading down the table) and a input fragment f (reading across the table), the fixed-point result is in cell $[b, f]$. Wide cells indicate that the same value appears in consecutive cells. The *pigeon-hole* artifact comes from the observation that for many pairs of values, the composited result doesn't change, *i.e.*, $OVER(f, b) = f$ or $OVER(f, b) = b$. More generally, for large volume where all voxel values are small, there will be a point at which new compositive values will not longer contribute to the final output image. For example, starting with a background value of zero and incoming fragment values of 1, the values produced will be: 0, 1, 2, 3, 4, 5, 6, 7, 8, 8, ... (repeating 8's forever). Using table 2, we start with

Foreground Value

Background Value $b\backslash f$	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	1	2	3	4	5	6	7	8		9	10	11	12	13	14	15
2	2	3	4	5		6	7	8	9	10	11	12		13	14	15
3	3	4	5		6	7	8	9		10	11	12	13		14	15
4	4	5		6	7	8		9	10	11		12	13	14		15
5	5	6		7	8		9	10		11	12		13	14		15
6	6	7		8		9	10		11		12	13		14		15
7	7	8		9		10		11		12		13		14		15
8	8		9		10		11		12		13		14		15	
9	9		10		11		12		13			14			15	
10	10		11			12			13			14			15	
11	11		12			13			14				15			
12	12			13				14					15			
13	13				14					15						
14	14						15									
15	15															
$b\backslash f$	00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15

Table 2: Example of compositing of for 4-bit fixed point values f and g. $OVER(f, b)$ is found in the above matrix at $[b, f]$.

$f = 1$ and $b = 0$, then we have $[0, 1] \Rightarrow 1$. Repeating until $b = 8$, we find that $[8, 1] \Rightarrow 8$;

Iteration

Foreground f	b	1	2	3	4	5	6	7	8	9	10+
1	0	1	2	3	4	5	6	7	8...		
2	0	2	4	5	6	7	8	9	10	11	12...
3	0	3	5	7	9	10	11	12	13...		
4	0	4	7	9	10	11	12	13	14...		
8	0	8	12	14	15...						
10	0	10	13	14	15...						
15	0	15...									

Table 3: An example of repeated composition of selected 4-bit fixed point values. Repeated compositions of the foreground values are show from left to right. Long cells with $\#\ldots$ show that repeated compositions no longer change its value.

Table 3 show this affect of a selected set of 4-bit values. Given a background value of 14, foreground values of 0 to 7 produce a value of 14, while foreground fragment values of 8 to 15 produce a value of 15; *i.e.*, half the input values will not change a value of 14 was stored in a 4-bit fixed-point frame buffer.

It is "common knowledge" in the visualization community that front-to-back compositing produces higher quality imagery than back-to-front. The *pigeon-hole* artifact describes this phenomenon quite clearly. Given a volume that, in aggregate, is fairly opaque, values at the end of a compositing run will contribute less (or nothing) to an image. Hence, starting at the front on the volume will allow the details closest to the viewer to contribute while details far away, which are likely to be occluded anyway, contribute less.

4.3 Weight rounding

This last artifact is more subtle and harder to characterize. The basic issue is that the OVER operator computes an affine combination of two fixed-point numbers, *i.e.*, $x = a \times t + b \times (1 - t)$. The relative opacities, t and $(1 - t)$ are affected very differently by rounding: the value t is typically very small, so the value $(1 - t)$ is large. For large datasets, the typical color values (a and b) are quite a bit larger than the per voxel opacity values (t). If the value t' is slightly larger than t, the total magnitude of the term $a \times t'$ is very small and it changes very little from $a \times t$, so is likely to round to the same value as $a \times t$, *i.e.*, $\lfloor a \times t \rfloor = \lfloor a \times t' \rfloor$. However, the total magnitude of $b \times (1 - t')$ is very large, and is not as likely to round to the same value as $b \times (1 - t)$, *i.e.*, $\lfloor b \times (1 - t) \rfloor > \lfloor b \times (1 - t') \rfloor$. Thus, in the case where $a = b$, a small increase in opacity produces a value that is actually smaller than the opacity for t, *i.e.*, $\lfloor a \times t + a \times (1 - t) \rfloor > \lfloor a \times t' + a \times (1 - t') \rfloor$. This artifact does not happen for every composition of a particular color and opacity value, but often enough to introduce significant error.

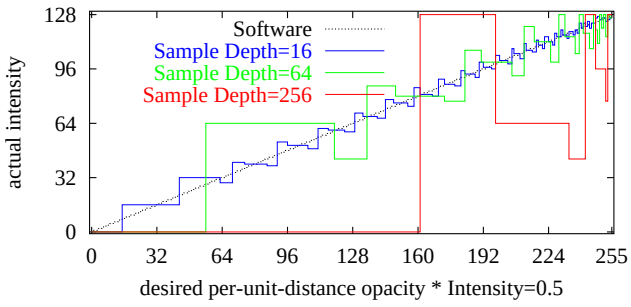


Figure 3: Plots of the "Weight rounding" artifact for sample depths of 16, 64, and 256.

We can see this affect in figure 3, with color=0.5 and per-unit-opacity varies linearly from 0.0 to 1.0, so the curves should follow a linear ramp from 0.0 to 0.5. The black line in figure 3 is software and linearly ramps from 0.0 to 0.5. As sample depth increases, the lines no longer strictly increase, but also decrease. The periodic decreases in the curves for sample depth of 16, 64, and 256, are those points at which the rounding down of $b \times (1 - t)$ occurs frequently, but the rounding up of $a \times t$ does not occur as frequently.

5 Improved Approximation by Dithering

Dithering is the process of producing a series of values that, in aggregate, produce a value that none of the values can represent directly. Our use of the term dithering is the same, except we only dither with respect to the composited values, or depth (Z) - we do no dithering in screen (X & Y) space.

If we wish to add a series of numbers that cannot be represented in an integer, say 1.5, we can choose either 1 or 2 to approximate it. The problem is that if we choose 1 and add it 10 times, the result is 10, not 15. Same if we use 2 - the result is 20. However, if we alternate by adding 1, then 2, and repeating, we obtain a result of 15. This is a simplistic example but illustrates the basic idea of the technique.

Our code affects the alternating of values by repeating a randomly permuted set values. For a dithering period of X, and a value of Y (with $f = Y - floor[Y]$, *e.g.*, f is the fractional part of Y), then we use $X - \lfloor X * f \rfloor$ terms of $\lfloor Y \rfloor$ and $\lfloor X * f \rfloor$ terms of $\lfloor Y \rfloor + 1$. For example, with a dithering of period 10, and a value of 1.4, on pattern that could be produced is:

$$\{1, 2, 1, 2, 1, 2, 1, 1, 2, 1, \ldots 1, 2, 1, 2, 1, 2, 1, 1, 2, 1, \ldots\}.$$

5.1 Implementation

Our experiments were conducted under OpenGL, and use palleted textures extension to implement dithering; that is, for a texture with an ordinal value of X which is meant to represent 2.5, one palette would translate it to a value of 2, while the other would translate it to a value of 3.

A two dimensional table is constructed at run time where the height corresponds to the size of the transfer function (256 entries, as we use just byte-sized scalar data) and the width is the dithering period. For each slice, we download the dither table corresponding to the slice number modulus the dithering period (*e.g.*, repeating the dithering pattern). This path is fully optimized in hardware on our system. However, one could use dependent texture lookups to implement this kind of dithering. We have not yet examined the performance issues of downloading a new palette each slice vs. the cost of dependent texture lookups.

We use a dithering period of 11 for most of our experiments, though we show in section 5.2 that using a dithering period of 256 can be necessary. There is a relationship of the smallest per-voxel value and the smallest value that a dithering pattern can capture that we have not yet fully explored, but it is clear that the size of the dithering period should be a function of the sample depth (*i.e.*, dataset depth).

5.2 Results

Our dithering technique can significantly improve image quality. However, our "round-up–round-down" dithering technique only really deals with the first artifact, representation. Our rounding technique only works on the pigeon-hole artifact if the foreground fragment value happens to be at the end of a run of similar values (*i.e.*, at the right end of a long cell in table 2). For example, if $OVER(x, y) = y$ and $OVER(x + 1, y) = y$, then dithering x with $\{x, x + 1\}$ will accomplish nothing. To deal with this, the dither increment must be larger than 1. However, OpenGL has no way to modify a background fragment by different amounts based on its value; nor is it practical to try to estimate a voxel's depth with respect to the screen in a general projection on a pixel-by-pixel basis (thereby estimating what the increment should be). However, a dither mechanism where the increment is something other that 1, based on the background fragment value, could be accomplished in a pixel program.

Spatial dithering has a problem in that thin or highly-transparent structures may be skipped entirely. For example, a series of 9 voxels along a ray have a value of 0.1. If the dithering period is less than 9, then all of the dithered values will be zero and no contribution is made. If the dithering period is 10, such that one voxel will be dithered to a value of 1, it is also possible for the voxels not to contribute if the time at which the rounding up occurs is just before or just after this set of 9 voxels. By extension, the appearant position of a boundary of some structures may move as a result of dithering. Given the last example of 9 voxels and a dithering period of 10: the position at which boundary appears will correspond to the position of the single dithering up of the voxel value.

(a) Software (b) Hardware (c) Dithering

Figure 4: Comparison of software, hardware, and hardware with dithering (sample depth = 64), with color=1.0.

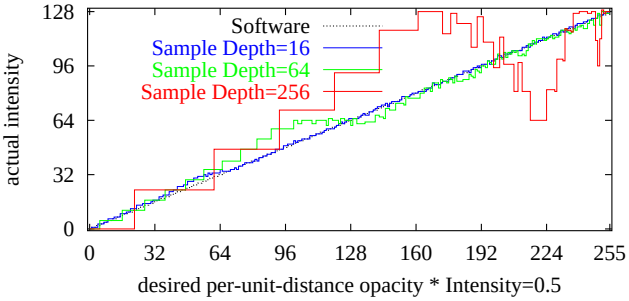


Figure 5: A comparison of software and hardware dithering for color=0.5. The dip in the dithering curves are due to the "weight-rounding" artifact.

The ramp dataset is shown in figure 4 shows that our dithering technique is able to recover most of the character of the software version (image 4(a)). Figure 5 shows the profile plots for the ramp dataset for color=0.5, using sampling depth of 16, 64, and 256. While the curve for sample depth of 16 tracks the software slope fairly closely, the curve for a sample depth of 64 has some noticeable wiggles and considerable over estimates the slope for the first half of the graph. The curve for a sampling depth of 256 shows that dithering is simply not capable of recovering the ramp. Our dithering technique can improve the image quality, but has trouble overcoming the "weight rounding" artifact.

Our second dataset is a Richtmyer-Meshkov (RM) turbulent-mixing simulation [MCC+99]. The dataset is $2048^2 \times 1980$ voxels by 237 timesteps, and we use the center 1024^3 of timestep 175. Figure 6 shows a series of renderings of the RM dataset: the images show a denser liquid on the right, moving into less dense liquid on the left. The color transfer function linearly maps density of 0.0 to green and 1.0 to red. The opacity transfer function maps the density to proportional desired opacity. The sampling depth is 1024 because we use 2D textures.

	Red			Green		
DP	L_∞	L_1	L_2	L_∞	L_1	L_2
1	195	-78.22	112.64	47	-20.01	23.21
4	67	-20.29	35.45	47	-20.01	23.21
16	68	-17.76	35.03	42	12.54	22.33
64	68	-16.31	35.49	49	13.43	22.80
256	68	-16.00	35.05	52	12.66	18.90

Table 4: Hardware dithering on RM dataset, time-step 175. This table shows error values for the red and green channels of the images in figure 6.

Figure 6 illustrates increasing the dithering period from 1 to 256 by powers of four. A dithering period one 1 is actually not dithering; no image is generated as all of the per-voxel values are too small to be represented and are simply truncated. A dithering period of 4 shows the essential outline of the high-density fluid in red on the right side and the entry of thin plumes of high-density fluid on the left. Orange appears in the right regions with a dithering period of 16. Dithering period 64 do not seem to add any important detail: there seems to be a lot of noise in the images. This can be largely attributed to the issue of dithering on thin, diaphanous structures: the noise results from the dithering pattern apparently moving the location of boundaries of these thin structures. At a dithering period of 256, we find the less dense liquid on the left finally appearing. While extremely faint, the value in the dithered image is the same as in the software image, 4 (on a scale of 0 to 255). Table 4 shows the L_∞, L_1, and L_2 errors for the red and green channels of the images in figure 6.

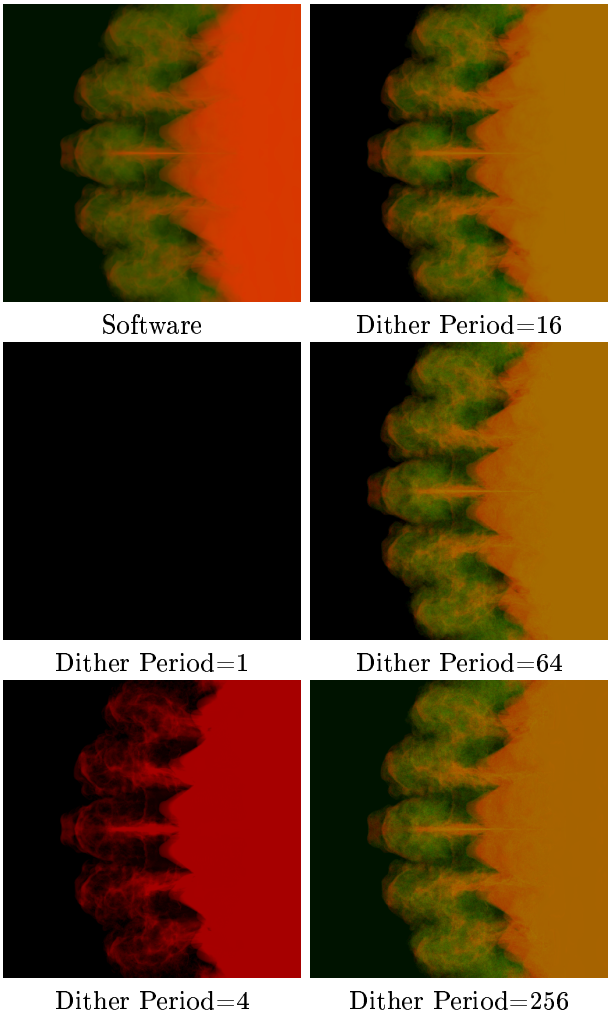


Figure 6: Comparing software vs hardware with dithering on RM dataset.

6 Exponent-factoring compositing

Our *exponent-factoring* technique comes from the observation that the opacity portion of the OVER operator can be factored into a hierarchy of operations where the exponent portion of the per-unit-to-per-voxel translation equation can be incrementally applied. The significance of this is that the intermediate per-voxel values can be many orders larger that the final per-voxel opacity values, which lessens the impact of limited width, fixed-point representations. For example, for a set of 32 slices, earlier techniques would compute a per-voxel opacity as $\alpha = 1 - (1 - \alpha')^{1/32}$, where alpha ($\alpha$) is not likely to representable in fixed-point. For example our technique could factor a set of 32 slices into a compositing tree with bifurcation factors associated with each level of the tree of $4 \times 2 \times 4$. The per-voxel opacity for the leaf level of the tree is calculated with a sample depth of 4, or $\alpha_{L=1} = 1 - (1 - \alpha')^{1/4}$. In the next level, the per-voxel alpha would be computed from the prior level's final alpha, or $\alpha_{L=2} = 1 - (1 - \alpha_{L=1})^{1/2}$; and the last (root) level would be $\alpha = \alpha_{L=3} = 1 - (1 - \alpha_{L=2})^{1/4}$. If we so

expand this series, we find:

$$\alpha = 1-(1-(1-(1-(1-(1-\alpha')^{1/4}))^{1/2}))^{1/4} = 1-(1-\alpha')^{1/32}. \quad (3)$$

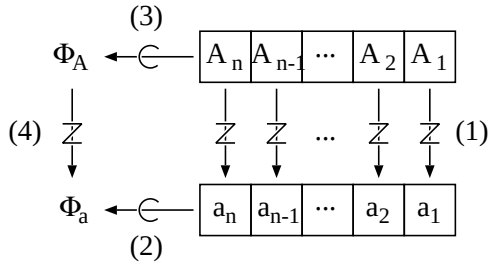


Figure 7: Two routes to the same result. 'Z' indicates "per-unit-distance to per-voxel opacity transformation" and 'C' indicates compositing. Normal compositing follows path (1) - (2); our technique follows path (3) - (4). A_i is per-unit-distance opacity, a_i is per voxel opacity, and Φ_a is the final, computed opacity. Φ_A is an intermediate opacity value.

While this technique can capture significantly more detail that dithering approaches, it cannot be used on the color channels because this factoring approach does not preserve the relative contribution of voxel colors. This technique could be used in applications where there is a strong need for visualizing large amount of highly transparent media. For example, this could be used to calculate highly accurate shadow volumes, occlusion, or lightmaps, or medical and industrial CT applications.

6.1 Proof

We wish to show that the alpha component of compositing is insensitive to the location/application of the fractional component of the unit-to-voxel-volume opacity relationship. See figure 7 for a graphical sequences of operations. Normal compositing follows steps (1) and (2): given per-unit-distance opacities A_i, compute the per-voxel a_i opacities, then composite this values to the final opacity ϕ_a. However, one can follow steps (3) and (4) to produce the same result: composites the per-unit-distance opacities A_i and produce an intermediate opacity of ϕ_A, then apply the per-unit-to-per-voxel opacity transformation to produce ϕ_a. Our inputs are the number of samples n and per-unit-distance opacity of sample i, A_i. The normal compositing computes the per-voxel opacity (see equation 1) for sample i :

$$a_i = 1 - (1 - A_i)^{\frac{1}{n}} \quad (4)$$

Then composites these values to produce the final opacity ϕ_a:

$$\Phi_a = 1 - \prod_{i=1}^{n} (1 - a_i)\,. \quad (5)$$

However, one can composite the per-unit-distance opacities first:

$$\Phi_A = 1 - \prod_{i=1}^{n} (1 - A_i) \quad (6)$$

Then apply the the per-unit-distance to per-voxel opacity calculation (*exponent-swizzle*, see equation 1):

$$\Phi_a = 1 - (1 - \Phi_A)^{\frac{1}{n}} \quad (7)$$

If we combine equations 4 and 5 (following path (1)-(2) from figure 7), we obtain:

$$\begin{aligned}
\Phi_a &= 1 - \prod_{i=1}^{n} (1 - \{a_i\}) \\
&= 1 - \prod_{i=1}^{n} \left(1 - \left\{1 - (1 - A_i)^{\frac{1}{n}}\right\}\right) \\
&= 1 - \prod_{i=1}^{n} \left((1 - A_i)^{\frac{1}{n}}\right) \\
&= 1 - \left(\prod_{i=1}^{n} (1 - A_i)\right)^{\frac{1}{n}}
\end{aligned}$$

If we combine equations 6 and 7 (following path (3)-(4) from figure 7), we obtain:

$$\begin{aligned}
\Phi_a &= 1 - (1 - \{\Phi_A\})^{\frac{1}{n}} \\
&= 1 - \left(1 - \left\{1 - \prod_{i=1}^{n} (1 - A_i)\right\}\right)^{\frac{1}{n}} \\
&= 1 - \left(\prod_{i=1}^{n} (1 - A_i)\right)^{\frac{1}{n}}
\end{aligned}$$

For both, we arrive at the formula, $\Phi_a = 1 - (\prod_{i=1}^{n} (1 - A_i))^{\frac{1}{n}}$, which shows that both paths are the same. This means that when we compute the per-voxel opacities from per-unit-distance opacities, we do not need to apply the full exponent all at once. If we did, we would get the situation shown in section 4.1 where the per-voxel values were simply truncated to zero for a large number of per-unit-distance values. As we showed in equation 3, this technique can be used to factor the per-unit-distance-to-per-voxel opacity transformation into a hierarchy of transformations.

6.2 Implementation

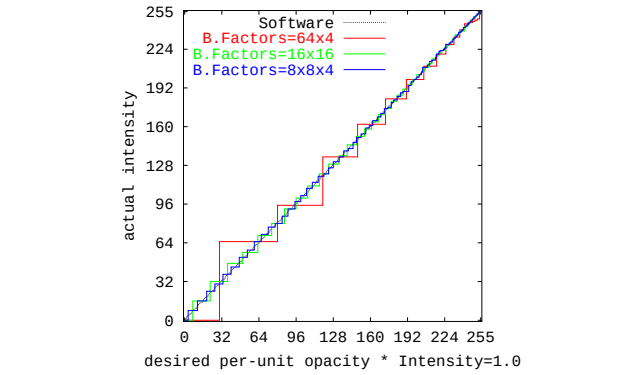
The current approach uses palleted textures to apply the initial transfer function and the PixelTransfer operations to apply the *exponent-swizzle* operation. The use of these features together can occasionally use non-optimized rendering paths. Implementing the swizzle operation in a dependant texture lookup should be considerably faster,

as these are highly optimized operations in new graphics cards. We plan to implement this so in future. The code routine is shown in figure Figure 8.

```
global Z: current slice in volume
begin display()
  call HierachicalRendering( root node )
  final image is in the frame buffer
end display
subroutine HierarchicalRendering( node )
  clear frame buffer
  if node is leaf
    download swizzle values for leaf nodes
    for each slice in leaf node
      composite slice Z
      increment Z
  else
    call HierarchicalRendering on all subnodes
    draw image from subnode's cache for all subnodes
  if node not root
    copy and swizzle image in buffer to node's cache
end HierarchicalRendering
```

Figure 8: The pseudocode for *exponent-factoring*.

6.3 Results

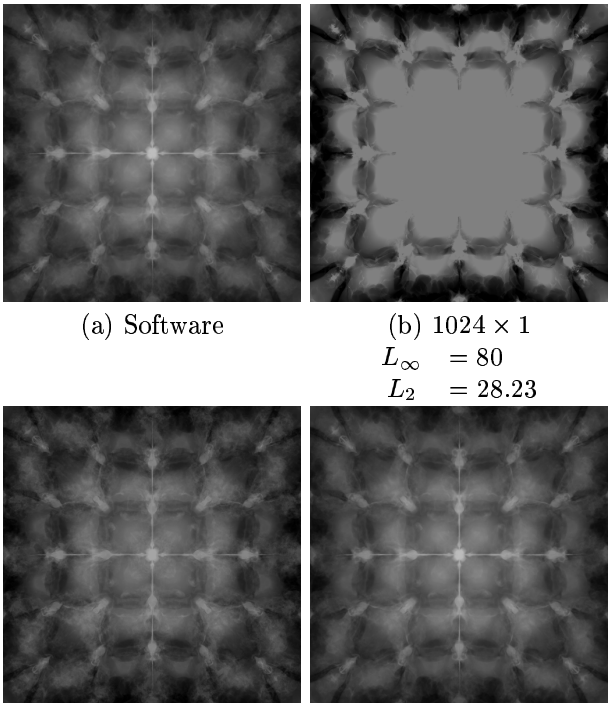


(a)

(b) B.F's=64x4 $L_\infty = 34$ $L_2 = 11.67$ (c) B.F's=16x16 $L_\infty = 9$ $L_2 = 2.9$ (d) B.F's=8x8x4 $L_\infty = 4$ $L_2 = 1.64$

Figure 9: The results of *exponent-factoring* on the ramp dataset, with a sampling depth of 256, and three different sets of bifurcation factors. The profiles of images (b), (c), and (d) are plotted in figure (a) in red, green, and blue, respectively. Compare these images with figure 2(d)

The results of this technique are quite encouraging. The error for the ramp dataset is considerably improved; we show three different combinations of bifurcation factors in figure 9. Notice that for the hierarchies of two



(a) Software (b) 1024×1 $L_\infty = 80$ $L_2 = 28.23$ (c) 256×4 $L_\infty = 28$ $L_2 = 9.70$ (d) $16 \times 8 \times 8$ $L_\infty = 4$ $L_2 = 1.19$

Figure 10: Comparing different set of bifurcation factors for the RM dataset.

levels (64×4 and 16×16), the error is much better when the two bifurcation factors are closer to each other. As discussed in earlier sections, the error is proportional to the exponent size, so minimizing the variance in the bifurcation factors minimizes the error associated with the tree. Also, as the tree height increases, the bifurcation factors decrease, and the error decreases.

The error for the RM dataset also considerably improved. We show images for three different combinations of bifurcation factors in figure 10. Figure 10(a) is rendered by software. Figure 10(b) is generated by compositing 1 group of 1024 slices (*i.e.*, no *exponent-factoring*; the naive hardware method). Figure 10(c) composites 4 groups of 256 slices. Figure 10(d) composites a hierarchy of 8 groups of 8 groups of 16 slices. For the hierarchies of height 2 the error decreases as the difference between the bifurcation factors in minimized. The error also decreases as the tree height is increased; a tree with bifurcation factors of 2 (and height=10) has $L_\infty = 2$ and $L_2 = 0.81$.

There are two parameters that affect the performance of this technique: the height of the compositing tree and the regularity of the bifurcation factors. The cost for compositing the original slices of the volume is fixed: each slice is only drawn once. However, each additional additional node in the compositing tree involves a buffer clear, compositing the cached images of the subnodes, copying the image from the framebuffer to a texture, and a application of *exponent-factoring* of the image. Increasing tree

height (and decreasing the bifurcation factors) decreases error but increases rendering time.

The OpenGL functions to perform the *exponent-factoring* operation are extremely slow and the current implementation as it is not useful for interactive viewing so we have not reported times for these tests. However, there are several places for improvement. *Exponent-factoring* could be performed using dependent texture lookups in a pixel program, the render-to-texture extension obviates image copy, and setting the maximum bifurcation factor to the number of allowable textures and performing all composite operations in a pixel program obviates a buffer clear.

7 Conclusions and Future Work

We have shown two techniques for improving rendering transparent objects on limited-width, fixed-point, graphics hardware. While both can significantly improve the quality and accuracy of a volume rendered image, there seems to be a limit to how much they can do. Dithering can capture color, but is not as accurate as *exponent-factoring*. It is also not one that can be applied indefinitely and still see improvement. *Exponent-factoring* can capture extremely small values but cannot capture color. Both techniques can be used in real-world applications, and have their place in the visualization toolbox.

Acknowledgments

This document was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor the University of California nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or the University of California. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or the University of California, and shall not be used for advertising or product endorsement purposes. This work was performed under the auspices of the U.S. Department of Energy by University of California, Lawrence Livermore National Laboratory under Contract W-7405-Eng-48.

References

[CCF94] B. Cabral, N. Cam, and J. Foran. Accelerated Volume Rendering and Tomographic Reconstruction Using Texture Mapping Hardware. In *Volume Visualization 1994*, pages 91–98, October 1994.

[EKE01] K. Engel, M. Kraus, and T. Ertl. High-quality pre-integrated volume rendering using hardware-accelerated pixel shading. In *Graphics Hardware 2001*, pages 9–16, August 12–13 2001.

[KWP01] K. Kim, C.M. Wittenbrink, and A. Pang. Extended Specifications and Test Data Sets for Data Level Comparisons of Direct Volume Rendering Algorithms. *IEEE Transactions on Visualization and Computer Graphics*, 7(4):299–317, October 2001.

[MCC+99] A.A. Mirin, R.H. Cohen, B.C. Curtis, W.P. Dannevik, A.M. Dimits, M.A. Duchaeineau, D.E. Eliason, D.R. Schikore, S.E. Anderson, D.H. Porter, P.R. Woodward, L. J. Shieh, and S.W. White. Very high resolution simulation of compressible turbulence on the IBM-SP system. In *Supercomputing 99*, November 13–19 1999.

[MHB+00] M. Meißner, J. Huang, D. Bartz, K. Mueller, and R. Crawfis. A practical evaluation of popular volume rendering algorithms. In *Volume visualization 2000*, pages 81–90, October 8–13 2000.

[PD84] T. Porter and T. Duff. Compositing Digital Images. In *Siggraph 1984*, pages 253–259, July 1984.

[Wal81] B.A. Wallace. Merging and Transformation of Raster Images for Cartoon Animation. In *Siggraph 1981*, pages 253–262, August 1981.

[WMG98] C.M. Wittenbrink, T. Malzbender, and M. Gross. Opacity-Weighted Color Interpolation for Volume Rendering. In *Volume Visualization 1998*, pages 135–142, October 19–20 1998.